

204
USE OF DEFINITE CLAUSE GRAMMARS

by

Weilin T. Chen

M.S. Kansas State University, KS, 1985

B.S. Providence College, Taiwan, 1980

A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1987

Approved by:

William J. Henkle
Major Professor

LD
215
.R4
MSC
1977
C83
C.2

Table of Contents

A11207 301458

Chapter One: Overview	1
1.1 Introduction	1
1.2 Organization	2
1.3 Results	3
Chapter Two: The Use of DCG in Prolog	4
2.1 The Structure of Prolog	4
2.2 Definite Clause Grammars	9
2.3 Examples of the Use of DCG	12
Chapter Three: Case Study	22
3.1 Overview	22
3.2 Selection of A Prolog Version	22
3.3 Sample Problem	24
3.4 Programs	28
3.5 Evaluations	30
3.5.1 Syntactical Complexity Measurement	30
3.5.1.1 Rules for the Complexity Measurement	30
3.5.1.2 Syntactic Measurement for the Implementations	36
3.5.2 Subjective Evaluation	40
3.6 Conclusions	42
Bibliography	44

Lists of Figures

Figure 2.1.1:	
A simple Prolog program defining "Sibling"	5
Figure 2.1.2:	
Resolution search of a goal in Fig. 2.1.1	6
Figure 2.1.3:	
Stack model of searching route for Fig. 2.1.2	7
Figure 2.1.4:	
Searching tree for Fig. 2.1.1	8
Figure 2.3.1:	
CFG for pattern 'A1 + A2 + ... + An'	12
Figure 2.3.2:	
DCG (step 1) for pattern 'A1 + A2 + ... + An'	13
Figure 2.3.3:	
DCG (step 2) for pattern 'A1 + A2 + ... + An'	14
Figure 2.3.4:	
DCG (step 3) for pattern 'A1 + A2 + ... + An'	15
Figure 2.3.5:	
DCG (step 4) for pattern 'A1 + A2 + ... + An'	17
Figure 2.3.6:	
Prolog program for pattern 'A1 + A2 + ... + An'	18
Figure 3.1:	
Distinct Predicates	31
Figure 3.2:	
Determining factors for syntactic measurement	32
Figure 3.3	
A simple computation example	34

Lists of Tables

Table 3.1:	
Grammar for predicate logic problem	25
Table 3.2:	
Totals for the example in Fig. 3.1	35
Table 3.3:	
Count results for the scanner	37
Table 3.4:	
Count results for the parser	38

Appendices

Appendix I: Grammars	47
Context Free Grammars	47
Definite Clause Grammars	49
Appendix II: C-Prolog Programs	52
Main Program (main.pro)	52
Database Program (pred.db.pro)	54
Scanner Program (pred.scan.pro)	56
Parser Program (pred.parse.pro)	58
Appendix III: Turbo Prolog Programs	63
Main Program (main.pro)	63
Database Program (database.inc)	69
Stacks Program (stacks.inc)	71
Scanner Program (scanner.inc)	72
Operators Program (operators.inc)	73
Appendix IV: Test Files	75
Appendix V:	
Formal Grammar for Definite Clause Grammars	76

Chapter One: Overview

1.1 Introduction

Prolog is a language for logic programming. It uses a backward reasoning, depth-first search algorithm for evaluating its goals. Because its rule structure is similar to grammar rules and its search algorithm is similar to the LL(1) parsing algorithm, Prolog is an excellent language for developing parsers by using grammar rules [Clo84] [Coh84] [Col85] [Ste86] [War80]. Several grammar rules have been introduced for parsing in Prolog. The most popular approach is the Definite Clause Grammar (DCG) [Klu85] [Per80]. Definite Clause Grammars are a generalisation of Context Free Grammars (CFG) in which the features are extended in two ways: the use of compound terms as arguments and the use of semantic condition expressions.

At present, the only documentation available for C-Prolog in the Computer Science Department at Kansas State University is the C-Prolog user's manual which does not cover the use of a DCG in Prolog. Thus, the first goal of this report is to establish both tutorial and user's references for the use of the DCG rule notations in C-Prolog. Secondly, this report presents a sample problem as an example to illustrate the use of a DCG for Prolog parsing.

Chapter Two of this report introduces the use of Definite Clause Grammars, it also presents several examples to illustrate the techniques of transforming a CFG to a DCG and a DCG to a C-Prolog program. To evaluate the use of a DCG, an implementation with the use of a DCG is written in C-Prolog. For

comparison, another implementation without the use of a DCG is done using Turbo Prolog. Both implementations are for the same problem domain— a form of the predicate logic.

Two different approaches for the evaluation of use of a DCG are reported. They are syntactical measurements and subjective evaluation. Syntactic measurement reports the total number of predicates, production rules, system calls, and called-predicates. A discussion of the computed syntactical complexity of both programs is given. The second approach is a subjective evaluation of the ease of program development and coding from the perspective of a programmer. A conclusion on the use of a DCG for parsing in Prolog is made based on the evaluation above.

1.2 Organization

Chapter Two presents the fundamental concept lying behind the use of a DCG for parsing. The first part gives a general idea about the structure of Prolog, and how Prolog relates to the parsing. The second part is an introduction to Definite Clause Grammars. Several examples are presented as illustrations of the use of a Definite Clause Grammar.

Chapter 3 is a case study. It explains the problem domain for the implementation project, the selection of Prolog tools, and the evaluation of the use of a DCG for Prolog parsing.

1.3 Results

DCG is an extension to a CFG. The DCG provides a good technique for the translation of a problem statement to a Prolog program. From the aspect of implementation design, the use of DCG for Prolog parsing is desirable despite the possibility of larger syntactical size than Prolog coding without the use of Definite Clause Grammars.

To provide users documentation about the use of a DCG for parsing in Prolog, a formal grammar (in BNF notation) defining the structure of a DCG is written. Furthermore, Section 2.3 of this report can serve as a tutorial lesson.

Chapter Two: The Use of a DCG in Prolog

2.1 The Structure of Prolog

A Prolog program is expressed as a collection of rules, facts, and goals. A rule is a clausal form with head and body (a sequence of subgoals), and it is written as:

$$A \text{ :- } B1, B2, B3, Bn. \quad (n > 0)$$

Without the body, the clause is called a fact, and a goal is the form without the head. The annotations are:

$$A. \quad (fact)$$

$$B1, B2, B3, Bn. \quad (goal)$$

The interpreter of Prolog is based on the resolution theorem prover [Rob65] using clauses to accomplish its goal [Klu85] [Clo84]. For example, Figure 2.1.1 is a simple Prolog program defining "sibling". Variable arguments begin with capital letters. Goal searching for "sibling" using resolution is shown in Figure 2.1.2.

```

1a) sibling(P,Q) :-
    P <> Q, parents(p(F,M1),P), parents(p(F,M2),Q).
1b) sibling(P,Q) :-
    P <> Q, parents(p(F1,M),P), parents(p(F2,M),Q).

2a) parents(p(joe,rose),tom).
2b) parents(p(joe,rose),dale).

Goal:
    sibling(tom,X).

```

Figure 2.1.1: A simple Prolog program defining "sibling".

Prolog starts with a goal statement "sibling(tom,X)" and searches for a clause whose head matches the goal (sibling(P,Q) of 1a in Figure 2.1.2), then unifies (binds) variable arguments as necessary to make matches succeed. The body of the instantiated clause is added to the goals to be satisfied, and Prolog tries to match this new goal to another clause, and so on. The matching mechanism described here is known as **resolution** [Ric83]. Each step of <I>, <II>, and <III> is a resolution. For instance, in step <I>, the resolution of two rules, *sibling(tom,X)* and rule 1a), results in a third rule

```

sibling(tom,X) :-
    tom <> X, parents(p(F,M1),tom), parents(p(F,M2),X).

```

In this example, the goal succeeds and unifies with "dale".

```

sibling(tom.X)
|
<I>  |
    1a)* | tom/P **
        | X/Q
        V

sibling(tom.X) :-
    tom <> X, parents(p(F,M1),tom), parents(p(F,M2),X).
|
<II> 2a) | F/joe, M1/rose
      |
      V

sibling(tom.X) :- tom <> X, parents(p(joe,M2),X).
|
2a) / joe/joe
    / M2/rose
    / X/tom
    V
tom <> tom ==> FAIL ***

2b) | joe/joe
    | M2/rose
    | X/dale
    |
    V
X=dale ==> SUCCEED

* "N)" indicates rule N) is used for resolution
** "Formal-var/Actual-var" indicates the variables unification
*** Backtracking occurs as search fails

```

Figure 2.1.2: Resolution search of goal in Figure 2.1.1

At any stage, if no matching fact or matching clause head can be found, Prolog will reject the most recently matched clause and undo any instantiation made in that match. Next, Prolog will reconsider the previous goal which matched the rejected clause, and try to resatisfy the goal. This attempt to find an alternative way to resatisfy previous goals is called backtracking. For example, in Figure 2.1.2, the backtracking occurs in step <III> as the searching fails (because a person cannot be a sibling of himself/herself). Thus, Prolog aborts the instantiation

of 2a), returns to the most recent backtracking point and tries the alternative (rule 2b).

Figure 2.1.3 shows the searching route for Figure 2.1.2 using a stack model. As each goal is expanded it links to its sequence of subgoals. Each goal is retained as a possible backtracking point, denoted with an "*". Each subgoal is expanded until it matches an existing fact.

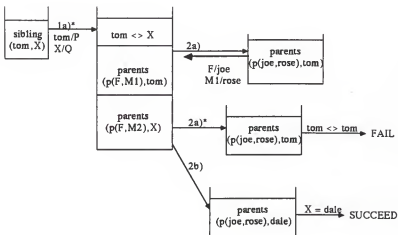
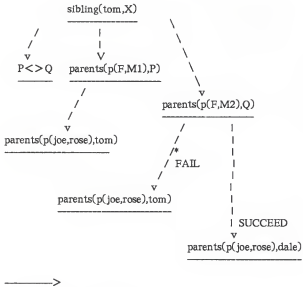


Figure 2.1.3: Stack model of searching route for Fig. 2.1.1

The searching routes of Figure 2.1.3 can be further expanded as a tree-like structure shown in Figure 2.1.4. It is noted that this entire Prolog performance uses depth-first strategies— top-down and left to right. The control structure of Prolog matches that of a recursive-descent top-down parser (LL(1) parser) [Aho86] [Bar86]. All of these indicate that Prolog uses a mechanical searching algorithm which enables Prolog to be an excellent language for parsing.



Note: * — backtracking point

Figure 2.1.4: Searching tree for Figure 2.1.1

2.2 Definite Clause Grammars

Context Free Grammars (CFG) is a fundamental class of grammars. The production of CFG is expressed as

$$\text{Head} \rightarrow \text{Body}$$

where the Head is a non-terminal and the Body is a sequence of terminals and/or non-terminals. CFG has some properties which are important to language parsing. It specifies the syntax of language in a concise and modular way. It also defines the embedded recursive structure of the language constructs [Aho86] [Bar86].

Definite Clause Grammars (DCG) are usually attributed to Pereira and Warren whose work was inspired by Colmerauer & Kowalski's idea [Col75] [Ste86]. The technique of DCG involves a simple generalisation of a CFG. DCG extends CFG annotations in two basic ways:

- 1) DCG allows non-terminals to have arguments. Also, an argument can be a compound term. A compound term has the form of

$$X(Y_1, Y_2, \dots, Y_i),$$

where $i \geq 1$. X is similar to the record name in Pascal-like language, and Y_i is the field. The structure of Y_i can be a number, a string, a list, or even another compound term. For instance, the expression,

`book(title,author,publisher(name,address),edition([1964,1970,1979,1987])), is`

a valid compound term.

The following is an example of using the compound term for non-terminal in a rule:

```
rel_exp(rop([Op,X1,X2]))  
    -> exp(X1), rop(Op), exp(X2).
```

The clause above defines a relational expression "rel_exp" as composed of a simple expression "exp", then a relational operator "rop" followed by another simple expression. The argument "rop([Op,X1,X2])" records the syntactic structure of the relational expression. It is the parse tree of "rel_exp". This extension technique is very useful in keeping track of construct quantities in the course of parsing.

- 2) In DCG, extra condition expressions are enclosed within a pair of curly brackets and remain in the right-hand side of the grammar rule. The expressions within the brackets are not involved in the underlying token consumption in the parsing. Rather, the extra condition provides some auxiliary conditions to restrict the constituents accepted. e.g.

```
digit_number(Number) ->  
    [Char], {Char > 48, Char < 60, Number is Char-49}.
```

The parse phrase of the example above consists of one component "[Char]"

only. The declarative semantics of the rule can be expressed as "the unification of `digit_number(X)` is true only if the parsed token, `Char`, is a digit of '0' to '9', then the corresponding integer is computed and reserved in the argument".

Occasionally, an extra condition is followed immediately after a particular parse phrase to perform a conditional test on the argument or token of that phrase. e.g.

```
lexpmore(Lop,L1,V1,Lf,Vf)
  -> [Lop], {Lop='eqv'}, lexp0(L2,V2),
      {evallog(V,V1,V2,eqv), Vf=V, Lf=lexp([eqv,L1,L2])}
```

In the example above, the semantic action `{Lop='eqv'}` is a conditional test for variable "Lop". The rest of the rule is processed only if the test succeeds.

These extra condition expressions are akin to the semantic rules (semantic actions) in attribute grammars [Coh85]. may have inherited and/or synthesized attributes [Aho86]. An example will be presented in the next section.

2.3 Examples of the Use of a DCG

The following example is to be used as an illustration for the use of a DCG throughout the entire section. Consider the notation,

$$A1 + A2 + A3 + \dots + An \quad (n > 0),$$

which can be interpreted as:

it is the sum of integers $A1, A2, \dots$, and An , or

it is the concatenation of strings $A1, A2, \dots$, and An .

Figure 2.3.1 is a Context Free Grammar for the pattern described above, and it parses the expressions, "*Art + Of + Prolog*" and "*2 + 4 + 6*", correctly.

```
expression -> sum_expression
expression -> string_expression

sum_expression -> int_variable more_sum_exp
string_expression -> str_variable more_str_exp

more_sum_exp -> '+' int_variable more_sum_exp
more_sum_exp -> null

more_str_exp -> '+' str_variable more_str_exp
more_str_exp -> null

int_variable -> 2
int_variable -> 4
int_variable -> 6
str_variable -> "Art"
str_variable -> "Of"
str_variable -> "Prolog"
```

Figure 2.3.1: Context Free Grammar for pattern ' $A1 + A2 + \dots + An$ '

In DCG, a terminal is enclosed within a pair of square brackets, such as `[+]` `== '+'`, `[2]` `== 2`, `[Pro]` `== "Pro"`, `[]` `== null`. A comma `,` is used as the delimiter if there is more than one literal in the body of the rule and a period is an end-mark for a production rule. With few changes in notations, the CFG in Figure 2.3.1 is transformed to a DCG which is listed in Figure 2.3.2. The modified grammar (step 1) is still a CFG but it is written with DCG notation.

```

expression -> sum_expression.
expression -> str_expression.

sum_expression -> int_variable, more_sum_exp.
string_expression -> str_variable, more_str_exp.

more_sum_exp -> [+], int_variable, more_sum_exp.
more_sum_exp -> [].

more_str_exp -> [+], str_variable, more_str_exp.
more_str_exp -> [].

int_variable -> [2].
int_variable -> [4].
int_variable -> [6].
str_variable -> ["Art"].
str_variable -> ["Of"].
str_variable -> ["Prolog"].

```

Figure 2.3.2: DCG (step 1) for pattern 'A1 + A2 + + An'

In Figure 2.3.2, the predicates `"sum_expression"` and `"string_expression"` have similar syntactic structures representing different semantics. To combine similar predicates as one, an argument (with parameter `Type`) is added to each non-terminal of the predicates above to reflect their semantic differences. The modified DCG is shown in Figure 2.3.3. This new grammar (step 2) is no longer

a CFG. It is context sensitive. The parameter "Type" is used to express the context constraint that the "variable" tokens all have the same "Type"s.

Note that the language which is recognized is still the same, but it is described by a smaller context sensitive DCG rather than a larger CFG.

```
expression -> variable(Type), more_exp(Type).

more_exp(Type) -> [+], variable(Type), more_exp(Type).
more_exp(Type) -> [].

variable(integer) -> [2].
variable(integer) -> [4].
variable(integer) -> [6].
variable(string) -> ["Art"].
variable(string) -> ["Of"].
variable(string) -> ["Prolog"].
```

Figure 2.3.3: DCG (step 2) for pattern 'A1 + A2 + + An'

In order to have the function of execution as well as parsing, the DCG in Figure 2.3.3 is extended to have extra conditions. An extra condition consists of a set of semantic expressions which were enclosed within a pair of curly brackets. The alternatives within semantic expressions can be separated by semicolon ";" characters.

To perform the computation (add or concatenate) for Figure 2.3.3, one needs to know the value of every variable. Also, the result of each operation should be saved for the later use. Therefore, a parameter reflecting the value of each variable is added to each non-terminal. The rewritten DCG for Figure 2.3.4 with the extension of extra condition is listed as follows:

```

expression(Result) -->
    variable(Type,Val1), more_exp(Type,Val2),
    { Val2=[], Result = Val1;
      Type = "integer", Result = Val1 + Val2;
      Type = "string", Result = Val1 cat Val2 }. **

more_exp(Type,Val) -->
    [], {Val = []}.
more_exp(Type,Val) -->
    [+, variable(Type,Val1), more_exp(Type,Val2),
    { Val2 = [], Val = Val1;
      Type = "integer", Val = Val1 + Val2;
      Type = "string", Val = Val1 cat Val2 }

variable(integer,2) --> [2].
variable(integer,4) --> [4].
variable(integer,6) --> [4].
variable(string,"Art") --> ["Art"].
variable(string,"Of") --> ["Of"].
variable(string,"Prolog") --> ["Prolog"].

** "cat" == concatenation of strings.

```

Figure 2.3.4: DCG (step3) for pattern 'A1 + A2 + + An'

The declarative semantics for the extra condition in predicate expression is read as:

```

IF it is a simple case (Val2=[]) THEN
    the result (Result) is the attribute of first variable (Val1).

ELSE
    IF the type of variable is an integer THEN
        compute the sum (Result) of two variables (Val1 and Val2).
    ELSE
        IF the type of variable is a string THEN
            compute the concatenation (Result) of two variables (Val1 and Val2).

```

As mentioned in the previous section, a DCG has the features of an attribute

grammar. The attributes *Val1*, *Val2* and *Type* of *Variable* in predicate expression are terminals whose values are supplied by the lexical analyzer, and the value of *Result* is computed from the values of its children nodes— *Val1* and *Val2*. Thus, *Type* (of the "variable" in predicate "expression"), *Val1*, *Val2* and *Result* are synthesized attributes. The *Type* in "more_exp" is an inherited attribute because its value depends upon the instantiation of the *Type* for "variable" in predicate "expression" [Aho85].

It is often useful for grammar rules to compute the parse tree of the recognized input stream. With the extension of the compound term nonterminal, DCG has the capability of providing structure information about the constructs at each node during parsing. In Figure 2.3.4, a compound term reflecting the parsing structure is added to each literal expression as a parameter.

Of course, one would not usually compute both the expression value and its parsing structure, but these are included for completeness of the example. More commonly an interpretation of an input stream (or value in the example) would be computed from the parse tree.

There are two cases for predicate "variable" in Figure 2.3.5. Case II is modified from Case I for the generalisation purpose.

```

expression(Result, expression(Tree)) ->
    variable(Type,V1,Tree1), more_exp(Type,V2,var(Tree1),Tree,Op),
    { V1=[], Result = Val2 :
      Type = "integer", Result = V1 + V2, Op = sum ;
      Result = V1 cat V2, Op = cat
    }.

more_exp(Type,Val,Lastree,Tree,Op) ->
    [], {Val = [], Tree = Lastree }.
more_exp(Type,Val,Lastree,Tree,Op) ->
    [+, variable(Type,V1,Tree1),
     more_exp(Type,Val2,[Lastree,op(Op),var(Tree1)],Tree,Op),
     { Val2 = [], Val = Val1 :
       Type = "integer", Val = Val1 + Val2 ;
       Val = Val1 cat Val2
     }

% Case I
% -----
variable(integer,2,integer(2)) -> [2].
variable(integer,4,integer(4)) -> [4].
variable(integer,6,integer(6)) -> [4].
variable(string,"Art",string("Art")) -> ["Art"].
variable(string,"Of",string("Of")) -> ["Of"].
variable(string,"Prolog",string("Prolog")) -> ["Prolog"].

% Case II
% -----
variable(integer,X,integer(X)) -> [X], {is_integer(X)}.
variable(string,X,string(X)) -> [X], {is_string(X)}.

```

Figure 2.3.5: DCG (step 4) for pattern "A1 + A2 + + An"

In Figure 2.3.5, the parameter **Op** in predicate **more_exp** is an attribute for the type of operation (such as **sum**, or **cat**). The DCG of step 4 is readily translated into Prolog notations. In C-Prolog, each symbol of "**->**", "**[]**", and "**{}**" has a corresponding system operator (with exactly the same notation). The "**integer**", "**string**", and "**sum**" appearing inside the extra condition are atomic strings in C-Prolog, where an atomic string can be expressed by itself along or

within a pair of single quotes. Since the strings Art, Of, and Prolog are constant, they must be quoted to distinguish from the variable terms. The "cat" operator is supported by adding an extra set of user-defined rules. The "is_integer()" and "is_string()" within extra conditions of predicate "variable" can be substituted by the system functions "integer()" and "atom()". The implemented Prolog program (excludes the scanner part) for the DCG (with case II) in Figure 2.3.5 is listed in the following:

```
% Main control
% -----

start :-
    reconsult('pred.scan.pro').    % file "pred.scan.pro" is in Appendix II
    action('test.tutor',user).    % "test.tutor" contains an input string

action(Infile, Outfile) :-
    see(Infile),
    tell(Outfile),
    write(' Input tokens '), nl,
    write(' ----- '), nl,
    tab(6),
    get0(Ch),
    scan(Ch, Wordlist),
    seen, nl, nl,
    write(' Computation '), nl,
    write(' ----- '), nl,
    exp(Wordlist, Result, Tree),
    write(' The computed result'), nl,
    write(' ==> '),
    write(Result), nl, nl,
    write(' Parse tree : '), nl,
    writetree(Tree, 6), nl,
    action(____) :- seen, told.
```

Figure 2.3.6: Prolog program for pattern 'A1 + A2 + + An'

(To be continued)

```

% Printer facility
% -----
writree([X|Tail],Depth) :-
    X=[], !;
    X=variable(_), nl, tab(Depth+2), write(X), tab(3), writree(Tail,Depth);
    X=operator(_), nl, tab(Depth+2), write(X), writree(Tail,Depth);
    nl, tab(Depth), write(' '), D1 is Depth+2, writree(X,D1),
    nl, tab(Depth+1), write(' '), D0 is Depth-2, writree(Tail,D0).
writree([],_).

% Parser
% -----
exp(Expression,Result,Tree) :- expression(Result,Tree,Expression,S).

expression(Result, Tree) -->
    var(Type,V1,Tr1), more_exp(Type,V2,variable(Tr1),Tree,Op),
    { V2 = [], Result = V1 ;
      Type = integer, Result is V1+V2, Op = sum ;
      cat(V1,V2,Result), Op = cat
    }.

more_exp(Type,Val,Ptree,Tree,Op) -->
    [+, var(Type,V1,Tr1),
     more_exp(Type,V2,[Ptree,operator(Op),variable(Tr1)],Tree,Op),
    { V2 = [], Val = V1 ;
      Type = integer, Val is V1 + V2 ;
      cat(V1,V2,Val)
    },
    more_exp(Type,Val,Ptree,Tree,Op) --> [], {Val = [], Tree = Ptree}.

var(integer,X,integer(X)) --> [X], {integer(X)}.
var(string,X,string(X)) --> [X], {atom(X)}.

cat(S1,S2,Scat) :- name(S1,L1), name(S2,L2),
    append(L1,L2,Lcat), name(Scat,Lcat).

append([],L,L).
append([X|L1],L2,[X|Lcat]) :- append(L1,L2,Lcat).

```

Figure 2.3.6: Prolog program for pattern 'A1 + A2 + + An' (Continued)

The following is a list of terminal output from Prolog execution of an input string The + Art + Of + Prolog + by + Shapiro [note].

```
% prolog
C-Prolog version 1.4
|?- ['tutor.pro'].
tutor.pro consulted 3080 bytes 0.733333 sec.
yes

|?- start.
pred.scan.pro reconsulted 2440 bytes 0.683333 sec.
```

Input tokens

The + Art + of + Prolog + by + Shapiro

Computation

The computed result
==> ArtofPrologbyShapiro

Parse tree :

```
(
  (
    (
      variable(string(Art))
      operator(cat)
      variable(string(of))
    )
    operator(cat)
    variable(string(Prolog))
  )
  operator(cat)
  variable(string(by))
)
operator(cat)
variable(string(Shapiro))
```

```
yes
|?- halt.
```

[Prolog execution halted]

note

- I. " ^ Z" is a control character for end-of-input in C-Prolog (see reference [Clo84])). Thus, " ^ Z" appends after the input string in test file "test.tutor". However, user may use any other character for end-of-input by substituting the Ascii-code 26 (" ^ Z") in scanner file with the code corresponds to the replacing character, for instance, Ascii 26 is replaced by 46 if a "." is used as a control character for end-of-input.
- II. It always has the same operator for any valid input string in the example of 'A1 + A2 + ... + An', the input

The + Art + of + Prolog + by + Shapiro

is equivalent to

((((The + Art) + Of) + Prolog) + by) + Shapiro).

Chapter 3: Case Study

3.1 Overview

The objective of this chapter is to illustrate the use of a Definite Clause Grammar in Prolog for parsing and execution of a sample problem. The domain of this sample problem is a form of the predicate logic presented in Gries [Gri83].

In this study, an interpreter for the sample problem is implemented in C-Prolog with the use of DCG. For comparison, another version without the use of DCG is done using Turbo Prolog. The evaluation of the use of a DCG involves the measurement of the program size (or program complexity) and the analysis of the Prolog implementation. The former is determined by the ratio of the sum of the user-defined function calls and the system calls over the number of predicates. In the latter evaluation is made in terms of the ease of program design and coding during the course of implementation from the perspective of a programmer. In addition, a new form of comparison (the implementation-similarity) between two Prolog versions is introduced and discussed.

3.2 Selection of A Prolog Version

Prolog exists in a number of different implementations, each with its own semantic and syntactic peculiarities [Wae86]. At present, there are four imple-

mentations available in the Computer Science Department at Kansas State University— C-Prolog, Turbo Prolog, Quintus Prolog (runs on ZEROX A.I. workstation 1186), and AAI Prolog (runs on Macintosh). All, but Turbo Prolog, support the Definite Clause Grammar notation. Because both Quintus Prolog and AAI Prolog were not fully installed and Turbo Prolog was used for other projects [Ven87] [Che87] at the time author started this study, C-Prolog was selected for the example with the use of DCG and Turbo Prolog was chosen as a comparative version which does not use the DCG.

C-Prolog runs on the UNIX operating system (BSD 4.3) for the VAX 11/780. It is consistent with the "standard version" described by Clocksin & Mellish (C&M describe a core Prolog developed at Edinburgh) [Clo84]. C-Prolog provides both a full set of arithmetic operators and a set of operators for dynamic structures, such as "functor()", "=", and "call()". It also supports the Definite Clause Grammar rule notations with operators "—>", "[]", and "{}" [Cpo86]. Because it is interpreted, C-Prolog has slower execution speed compared to most of the conventional languages.

Turbo Prolog, released by Boland International, runs on MS-DOS for IBM-compatible Personal Computers [Bor86]. It provides several convenient facilities, such as a user-friendly interface system, full-screen editing/tracing, graphics/windowing capabilities and string handling. In addition, Turbo Prolog provides a typing system. Users are required to define the types and domains just as they do in Pascal and C. This feature not only provides the compiler with error-checking capabilities but also improves the performance efficiency. On the

other hand, Turbo Prolog can not dynamically create a structure due to its static typing system. The typing system and the lack of DCG built-in predicates keep Turbo Prolog, unlike other implementations, far away from the C&M standard. (Note: Recently, Boland International released a software package "Turbo Prolog Tool Box" which includes a preprocessor of DCG rule notations. Another similar translator is implemented by Watson [Wat87]).

3.3 Sample Problem

The problem domain for the Prolog implementation is a form of the predicate logic described by Gries [Gri83]. Table 3.1 shows the source grammar (in Backus-Naur Form) defining this sample domain.

$\langle \text{pred} \rangle ::= \langle \text{simpred} \rangle [\langle \text{lop} \rangle \langle \text{simpred} \rangle]^*$
 $\langle \text{simpred} \rangle ::= "(\langle \text{quant} \rangle)^* | \langle \text{lexp} \rangle$
 $\langle \text{quant} \rangle ::= \text{"forall" } \langle \text{quantbody} \rangle | \text{"thereexist" } \langle \text{quantbody} \rangle |$
 $\quad \text{"("numberof" } \langle \text{quantbody} \rangle \text{ ") " } \text{"=" } \langle \text{const} \rangle$
 $\langle \text{quantbody} \rangle ::= \langle \text{var} \rangle \text{" : " } "(\langle \text{ivar} \rangle \text{" .. " } \langle \text{ivar} \rangle \text{") " } \text{" : "}$
 $\quad \langle \text{avar} \rangle \text{" [" } \langle \text{var} \rangle \text{"] " } \langle \text{relop} \rangle \langle \text{const} \rangle$
 $\langle \text{lexp} \rangle ::= \text{"not" } \langle \text{lexp} \rangle | "(\langle \text{lexp} \rangle)^* | \langle \text{rexp} \rangle$
 $\langle \text{lop} \rangle ::= \text{"and" } | \text{"or" } | \text{"xor" } | \text{"imp" } | \text{"eqv"}$
 $\langle \text{rexp} \rangle ::= \langle \text{iexp} \rangle | \langle \text{exp} \rangle \langle \text{relop} \rangle \langle \text{exp} \rangle$
 $\langle \text{iexp} \rangle ::= \langle \text{ivar} \rangle \text{" in " } "[\langle \text{item} \rangle \text{" .. " } \langle \text{item} \rangle \text{"] "}$
 $\langle \text{item} \rangle ::= \langle \text{const} \rangle | \langle \text{ivar} \rangle$
 $\langle \text{exp} \rangle ::= \langle \text{var} \rangle | \langle \text{aexp} \rangle$
 $\langle \text{relop} \rangle ::= \text{"=" } | \text{">" } | \text{"<" } | \text{"<=" } | \text{">=" } | \text{"<=" }$
 $\langle \text{aexp} \rangle ::= \langle \text{avar} \rangle \text{" [" } \langle \text{item} \rangle \text{"] "}$
 $\langle \text{const} \rangle ::= \{ \text{any legal integer} \}$
 $\langle \text{ivar} \rangle ::= \{ \text{any defined index variable} \}$
 $\langle \text{var} \rangle ::= \{ \text{any defined variable} \}$
 $\langle \text{avar} \rangle ::= \{ \text{any defined array variable} \}$

Table 3.1: Grammar for predicate logic problem (To be continued)

Notes:

- I <term> is a nonterminal.
 "term" is a terminal or constant.
 {term}* == term has zero or more occurrences.
- II The precedence order for logical operator is
 "and" > "or" > "xor" > "imp" > "eqv".
 This order can be handled by grammars or by implementation itself.
- III All variables names like <ivar>, <var>, and <avar> are those defined and
 used in the problem statement.

Table 3.1: Grammar for predicate logic problem (Continued)

The predicate <pred> is defined as a simple predicate <simpred>, or it is defined as a <simpred> followed by more <simpred>s, where every two simple predicates are connected by a logical operator. The precedence of logical operators is and > or > xor > imp > eqv. The latter defines the predicate of a complex form. A simple-predicate <simpred> can be a quantification <quant> or a relational expression <lexp> associated with a "not" or a relational operator, e.g.

Sample predicates:

I> ("not" m), (m = xyz), (m < xyz), (m > xyz)

II> Universal Quantification
($\forall i : m < i < n : \text{array}[i]=0$)

III> Existential Quantification
($\exists i : m < i < n : \text{array}[i]=0$)

IV> Numerical Quantification
($(\sum_{i : m < i < n : \text{array}[i]=0}) = 50$)

The following are examples of compound predicates:

1. $((x-y > z) \text{ or } (m = z))$
2. $(z \text{ in } [m..n]) \text{ imp } ((z > x-y) \text{ and } (z < x+y))$
3. $((m = 0) \text{ and } (n = 10)) \text{ and } (\forall i : m < i < n : \text{array}[i]=0)$

In order to exercise the Prolog execution for the sample problem, a database file and several test case files are created. All the variables to be used in the test file are pre-defined and/or pre-assigned constant-values in the database file. For example, in the database, a 10-cell array byte has binary values as 00001111, and each variable of x, y, and z was bound with value 1, 2, and 3, respectively. Then, with the pre-defined variables described above, the following three syntactically-valid inputs are parsed and evaluated (the value TRUE/FALSE is the computed result):

- (1) forall i:(x..z):byte[i]=0
TRUE
- (2) (x>1) or (y>x) and (y<z)
TRUE
- (3) (y=x) and (byte[y]=0)
FALSE

3.4 Programs

The source grammar (in BNF notation) of the Turbo Prolog implementation is listed in Table 3.1. The source grammar was transformed into a Definite Clause Grammar for the C-Prolog implementation.

The following is a list of grammars and programs which are involved in the implementation for the sample problem described in the previous section. A brief description about the functions of each program is included within the parentheses.

Grammars

Source Grammar (in BNF notation)
Definite Clause Grammar

Table 3.1
Appendix I

Programs

C-Prolog:

Appendix II

Main Program
(Main Control; Menu Establishment)

Database Program
(Variables/Constants Assertion and Retraction)

Scanner Program
(Lexical Analysis)

Parser Program
(Parsing, Evaluation)

Turbo Prolog:

Appendix III

Main Program
(Main Control; Parsing and Evaluation)

Database Program
(Variables/Constants Assertion and Retraction)

Stacks Program
(Simulation of a Stack— Pop/Push a value)

Operators Program
(Evaluation of Relational/Logical Expression;
Determination of Precedence Order of the Logical Operators)

Scanner Program
(Lexical Analysis)

3.5 Evaluations

3.5.1 Syntactical Complexity Measurement

3.5.1.1 Rules for the Complexity Measurement

For the syntactic complexity measurement, Prolog clauses can be grouped into three categories:

RULES: A RULE has a complete clausal form, "Head :- Tail". The "Head" is always associated with an argument.

CONTROL RULES:

A CONTROL RULE has the same form as that for a RULE, except the "Head" is not associated with any argument. Since a control rule is invoked by calling its predicate name without any restriction (or parameter), it is used for the purpose of control, e.g. start :- openfile(F), closefile(F).

FACTS: A FACT consists of only the left-hand side of a general clausal form; that is, a FACT is the "Head" part of a RULE.

In Prolog, a distinct predicate is a unique name of the "Head" shared by one or more facts/production rules. Of course, a goal is also regarded as a distinct predicate. For instance, Figure 3.1 shows two distinct predicates. Predicate getspecialsymbol has a set of four facts in which each has a different value for its argument. Predicate const has three production rules, where each production rule

has discrete constituents for its "Tail".

```
getspecialsymbol(40, '(').  
getspecialsymbol(41, ')').  
getspecialsymbol(91, 'I').  
getspecialsymbol(93, 'J').
```

a) Predicate getspecialsymbol.

```
const(num(X),X) -> [X], {number(X)}.  
const(var(X),Y) -> [X], {var(X,Y1), Y=Y1}.  
const(ivar(X),Y) -> [X], {ivar(X,Y1), Y=Y1}.
```

b) Predicate const

Figure 3.1: Distinct Predicates

It is noted that the syntactical complexity (or the size) of a predicate is dependent upon the number of the facts, or it is determined by the number of the production rules, the type of the component, the number of occurrences for each component, and the parameters used for each occurrence in the "Tail". Therefore, all of the items described above should be taken into consideration for the complexity measurement of a program. Figure 3.2 is a list of five major items used in the computation of the complexity measurement.

- I) the number of distinct predicates.
- II) the number of production rules or facts.
- III) the number of called-predicates in the "Tail".
- IV) the number of system and I/O calls in the "Tail".
- V) the number of parameters used in the function calls on the "Tail".

Notes:

- a) "Tail" is the right-hand side of the general clausal form, Head :- Tail.
- b) A "called-predicate" is also a predicate.

Figure 3.2: Determining factors for the complexity measurement.

The factor of item II in Figure 3.2 varies with the categories to which it applies. For instance, item II is the number of production rules if applied to categories RULES or CONTROL RULES, and it is the number of facts to category FACTS. In addition, because the last three items in Figure 3.2 are particularly involved in the "Tail" of a rule, they are not applicable to the category FACTS.

As described in the previous section, there exists some syntactical differences, especially the built-in predicate notations, between C-Prolog and Turbo Prolog. In order to resolve the problem of notational difference and to maintain the consistency of the measurement, there is a need to standardize the counting unit of each item described in Figure 3.2. The following are the basic rules used to determine the count of system (I/O) calls or the number of parameters used in a procedure call:

- (1) A built-in predicate or a system operator is a system (I/O) call, e.g. write, read, call, frontlist, not, nl, fail, "=", "=>", and cut "!".
- (2) The special notations for DCG grammars in C-Prolog are system calls, such as "=>", "[]", and "{ }".
- (3) A ";" is a system call if it is encountered in an extra condition; otherwise, it is regarded as the control of processing an alternative production rule or a fact. For instance, in the following rule for parsing a letter,

is_letter(X) -> [X], {X>64,X<91; X>96,X<123}.

if variable X fails in the first condition-test for a capital letter, the alternative inside the "{}" is taken, which is a condition-test for a small letter, rather than an alternative of this production rule.

- (4) A compound term expression or a list expression is counted as a singular form. For instance, the expression "arith_exp"

arith_exp(multiply(*),3,[5,plus(+),7])

has 3 parameters. The arguments multiply(*), 3, and [5,plus(+),7] are a compound term, a number, and a list, respectively. More illustrations are to be given later in the computation example.

- (5) Anonymous parameters, denoted as underscore characters, appearing consecutively in the right part of an argument list can be waived from the parameter-counting. For example, the procedure call `universal(_Low,High,_)` is counted as three parameters, the first underscore "`_`" Low, and High.

To illustrate the rules described above, a small section from C-Prolog parser implementation, is given as follows:

```
(a) exp([minus,L1,L2,Val]) ->
    S.

    const(L1,V1), ['-', exp(L2,V2,_), {Val is V1+V2}.
    P.  A.  A.  S.A.  P.  A.  A.  S.A.  S.  A.

(b) exp([array,avar(X),ivar(L)],Val) ->
    S.

    [X], ['[', exp(L,V1,_), [']'].
    S.A. S.A.  P.  A.A.  S.A.

    {avar(X), Val=V2, Goal =., [X,V1,V2], call(Goal)}.
    S.P.  A.  A. S.A.  A.  S.  A.  S.  A.
```

Notes: The capital letters appearing under the rule notations indicate the type of count (or the item) notation is considered. The items the capitals represented are:

P. — Called-Predicate
 S. — System/IO call
 A. — Argument/Parameter

Figure 3.3: A simple computation example

In Figure 3.3, predicate *exp* has two production rules, (a) and (b). By applying rules (1) and (2), each $/X/$ or *call(Goal)* is counted as a system call associated with one parameter. The system call *Val is V1+V2* is considered as one system call with two parameters because its operation sequence can be written as *is(Val,+(V1,V2))*, where the $+(V1,V2)$ is a compound term. The list $/X,V1,V2/$ appeared in the expression *Goal* $\leftarrow \dots /X,V1,V2/$ is counted as one parameter according to rule (4). The resulting totals for the example above are listed in Table 3.2.

Items:	(I)	(II)	(III)	(IV)	(V)
Rule (a)			2	4	7
Rule (b)			2	8	11
Total	1	2	3	11	18

Table 3.2: Totals for the example in Figure 3.1

3.5.1.2 Syntactic Measurement for the Implementation

The functions of both Prolog programs include lexical analysis, syntactical analysis, parsing, and computation. Since the parser is the major subject of interest for this report, and it is easier to compare the complexities of two programs that function closely, both Prolog implementations were rearranged into two parts for the complexity measurement. They are the scanner part the parser part.

In C-Prolog, the files "main.p" (excluding the "menu" function) and "pred.scan.p" are combined as the scanner part, and the file "pred.int.p" stands itself as the parser part. For Turbo Prolog, the parser includes the "parser" section of the file "main.pro" and the file "operator.inc". The rest of the file "main.pro" and the file "scanner.inc" compose the scanner part. Since only existential quantification was fully implemented in Turbo Prolog, only existential quantification is included in the quantification section for both parsers.

Tables 3.3 - 3.4 list the count results of the scanner and the parser for both C-Prolog and Turbo Prolog implementations.

Items —>		# of individ.	# of produc.	# of	# of	# of
Categories		pred.	rules/ facts	called pred	system & I/O calls	para.
I						
I						
V		(I)	(II)	(III)	(IV)	(V)
<hr/>						
RULES	T.I	7	24	21	33	66
	C.I	8	20	19	70[a]	87[a]
<hr/>						
CONTROL RULES	T.I	1	1	5	3	7
	C.I	2	2	3	10	8
<hr/>						
FACTS	T.I	0	0	~	~	~
	C.I	1	11[a]	~	~	~[m]

[a]: a discussion is made in the context.

[m]: ~ = Not Applicable.

T. = Turbo Prolog

C. = C-Prolog

Table 3.3: Count results for the scanner.

Referring to Table 3.3 (the result for the scanner part), it is noted that the C-Prolog version has higher count of system calls (item V). This is because C-Prolog does not support string I/O handling as Turbo-Prolog does. In C-Prolog, each input is fetched on the character-by-character basis using system predicates, such as `get()`, `get0()`. Besides, the user is required to explicitly manipulate both the recognition of an input character and the type conversion from characters to strings. Therefore, both the count of the facts (item II) and the count on the parameter-passings in category RULES (item IV) are slightly higher in C-Prolog.

Items —>		# of	# of	# of	# of	# of
Categories		individ.	produc.	called	system &	para.
		pred.	rules/	pred	I/O calls	
			facts			
V		(I)	(II)	(III)	(IV)	(V)
RULES	T.I	20	79	81	148	322
	C.I	19	60	54	216[b]	344
FACTS	T.I	1	4	~	~	~
	C.I	0	0	~	~	~[m]

[b]: a discussion is made in the context.

[m]: "~" = Not Applicable.

T. = Turbo Prolog.

C. = C-Prolog.

Table 3.4: Count results for the parser.

In Table 3.4, C-Prolog has lower counts on the Production Rules and the Called-Predicates (items II and III). However, parsing with DCG rule notations invokes higher number of system calls because most of the rule notations are the built-in functions, such as "[]" etc.

Section 3.5.1.1 mentions that the syntactical complexity of a predicate is determined by the number of production rules, and the type, the number of the constituents in the "Tail". Considering the implementation in the parser part, if the occurrence of both a Called-Predicate and a System Call are given the same complexity weight, then the syntactical complexity of a predicate can be expressed as an equation:

$$\frac{(\text{count of System Calls} + \text{count of Called-Predicates})}{\text{count of Production Rules}} \quad (\text{Eq. 1})$$

Since the number of Production Rules can be varied by minor editing while the function of a predicate is preserved, the denominator of the equation above is replaced by the count of the Predicates.

$$\frac{(\text{count of System Calls} + \text{count of Called-Predicates})}{\text{count of Predicates}} \quad (\text{Eq. 2})$$

According to Equation 2, the calculated complexity values for both parser are:

$$\text{Turbo Prolog: } \frac{81 + 148}{20} = 11.5$$

$$\text{C-Prolog: } \frac{54 + 216}{19} = 14.2$$

The results indicate that on average C-Prolog has a larger size or a more complex syntax per Predicate in the parser implementation.

3.5.2 Subjective Evaluation

This section presents a different comparison of the two Prolog implementations. The comparison is made and discussed in term of the ease of learning and program development during the course of the implementation for this project. Most of the discussions will be focused on the parser implementation.

For a Prolog beginner who is familiar with conventional languages (like C and Pascal), Turbo Prolog would be an easier version for learning Prolog programming. That is because Turbo Prolog provides type-checking which helps point out syntactical problems. The trace function is also a plus feature to the beginner. As the trace mode is on, the cursor follows along from rule to rule in the editor window in addition to the output of the trace message. It shows exactly how an instantiation occurs at every step during searching. In addition, the *Turbo Prolog Owner's Handbook [Bor86]* provides the user with an excellent aide in exercising the fundamental programming skills in Prolog.

C-Prolog, on the contrary, is often frustrating to users during the learning stage because of its dynamic typing system and its lack of proper documentation. Generally, it takes a user more effort in learning C-Prolog programming, especially the use of DCG rule notations for parsing. However, as users become more familiar with the C-Prolog (DCG) notations, they are able to implement parsers in a more convenient way.

Parsing in Turbo Prolog, on the other hand, involves more user effort, such as the transformation from the source grammar to the program. In order to

achieve the same performance as that in C-Prolog parser, Turbo Prolog users need to define more predicates and invoke more predicate-calls. However, if any error occurs, the tracer does not give adequate help in debugging in spite of the user-friendly feature for Prolog beginner described earlier. That is because the tracer does not provide the depth-level information at each unification/backtracking point during searching.

Let "implementation-similarity" be the degree of closeness (or similarity) in syntax and structure of a Prolog parser to its source grammar. Higher implementation-similarity of a Prolog program indicates the structure/syntax of the parser is closer to its source grammar. It also implies that it requires less effort for a Prolog programmer to do the program development.

Referring to C-Prolog and Turbo Prolog parser programs (Appendixes II, III) and their source grammars (Table 3.1 and Appendix I), it is observed that C-Prolog version has a much higher implementation-similarity than Turbo Prolog does. In other words, the C-Prolog program preserves most of the structures or the syntax from its source grammar (DCG). Because the DCG provides a straightforward, less error-prone technique for the translation, the user is able to implement a program with the features of higher integrity, readability and correctness.

3.6 Conclusions

C-Prolog is consistent with the C&M Prolog version. It supports both the DCG rule notations and dynamic data structures. Turbo Prolog provides a typing system for the compiler and supports several user-friendly facilities. However, Turbo Prolog does not have built-in DCG rule notations, and it also lacks the flexibility in the dynamic creation of rule structures.

The syntactical complexity measurement presented in Section 3.5.1 draws some interesting results. With the support of string I/O manipulations, Turbo Prolog is more advantageous over C-Prolog for scanner implementation. In the C-Prolog parser, the syntactical structure of each production rule has a slightly higher complexity despite the use of DCG rule notations.

Programmers may feel more comfortable in Turbo Prolog programming during the earlier stage (learning stage) because of Turbo Prolog's user-friendly facilities and good tutorial reference. However, C-Prolog has much higher implementation-similarity, which indicates the higher similarity between of a C-Prolog program and its source grammar. Because Prolog implementation using DCG requires less program development in the transformation, users can write a parser program with higher integrity, readability, and correctness.

To conclude, DCG provides a good tool for parsing in Prolog. From the aspect of implementation design, the use of DCG for Prolog parsing is desirable despite the possibility of higher syntactical complexity.

To provide other users documentation about the use of a DCG for parsing in

Prolog, a BNF defining the structure of a DCG is listed in Appendix II. In addition, the examples of Section 2.3 serve as a tutorial reference.

BIBLIOGRAPHY

[Aho86]

Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Bar86]

Barrette, W.A., Bates, R.M., Gustafson, D.A., and Couch, J.D., *Compiler Construction, Theory and Practice*, Science Research Associations, Inc., Chicago, 1986.

[Bor86]

Turbo Prolog Owner's Handbook, Borland International, 1986.

[Bra86]

Bratko, Ivan., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1986.

[Bur85]

Burnham, W.D., and Hall A.R., *Prolog Programming and Applications*, Macmillan Education Ltd., London, 1985.

[Che87]

Chen, Spencer Shu-Tsu, "Prolog Implementation of a Graphic Tool for Generation of Ada Language Specifications," Master's Report, Kansas State University, Manhattan, KS, 1987.

[Clo84]

Clocksins, W.F., Mellish, C.S., *Programming in Prolog*, Springer-Verlag, New York, 1984.

[Coh85]

Cohen, Jacques "Describing Prolog by Its Interpretation and Compilation," *Communications of the ACM*, 28, pp1311-1324, 1985.

[Col75]

Colmerauer, A. Les grammaires de metamorphose. Groupe d'Intelligence Artificielle. Univ. of Marseilles-Luminy, France, 1975. (Appears as Metamorphosis grammars In *Natural Language Communication with Computers*, L., Bale, Eds., springer-Verlag, New York, 1978, pp.133-189.)

- [Col85]
Colmerauer, Alain, "Prolog in 10 Figures," *Communications of the ACM*, 28, pp1296-1310, 1985.
- [Cpo86]
C-Prolog User's Manual Version 1.4, Edited by Fernando Pereira, SRI International, Menlo Park, CA, February 7, 1986.
- [Gen85]
Genesereth, M.R., and Ginsberg M.L., "Logic Programming", *Communications of the ACM*, 28, pp.933-941, 1985.
- [Gri83]
Gries, D., *The Science of Programming*, Spring-Verlag, New York, 1983.
- [Klu85]
Kluzniak, F., and Szpakowicz, S., *Prolog for Programmers*, Academic Press, Orlando, 1985.
- [Lau86]
Lauxman, S. R., "The Design of Three Interpreters: Proposition, Proposition Proof, & Predicate," Master's Report, Kansas State University, Manhattan, KS, 1986.
- [Mun86]
Munakata, T., "Procedurally Oriented Programming Techniques in Prolog," *IEEE EXPERT*, Summer 1986, pp.41-47.
- [Per80]
Pereira, F. C. N., and Warren D. H. D., "Definite Clause Grammars for Language Analysis- A survey of the Formalism and a Comparison with Augmented Transition Networks," *Artificial Intelligence* 13 (1980), 231-278.
- [Ric83]
Rich, Elaine, *Artificial Intelligence*, McGraw-Hill, New York, 1983.
- [Rob65]
Robinson, J. Alan, "A Machine-Oriented Logic Based on the Resolution Principle" *J. ACM* 12(1), pp.23-41.
- [Rub86]
Rubin D., "Turbo PROLOG: A PROLOG Compiler for PC Programmer" *AI EXPERT*, Premier 1986, pp.87-97.

- [Ste86]
Sterling, L., and Shapiro, E., *The Art of Prolog; Advanced Programming Techniques*, MIT Press, Cambridge, MA, 1986.
- [Sub85]
Subrahmanyam, P.A., "The "Software Engineering" of Expert Systems: Is Prolog Appropriate?" *IEEE SE-11* , No.11, pp.1391-1400, 1985.
- [Ven87]
Venkatech, K.S., "Interactive Tehniques for a Program Generator Using Prolog." Master's Thesis, Kansas State University, Manhatta, KS, 1987.
- [War80]
Warren, David H.D., "Logic Programming and Compiler Writing," *Software Practice and Experience* 10, pp87-125, 1980.
- [Wat87]
Watson, James F., "A Grammar Rule Notation Translator" *SIGPLAN Notice* 22, #4, pp16-27, April 1987.
- [Wee86]
Weeks, J., and Berghel, H., "A Comparative Feature-Analysis of Microcomputer Prolog Implementations," *SIGPLAN Notices* 21, #2, February 1986, pp.46-61

Appendix I: Grammars

Context Free Grammar

Interp \rightarrow Pred MorePred

Morepred \rightarrow Lop Pred Morepred

Morepred \rightarrow null

Pred \rightarrow "(" Quant ")"

Pred \rightarrow Lexp0

Quant \rightarrow "forall" Quantbody

Quant \rightarrow "thereexist" Quantbody

Quant \rightarrow "(" "numberof" Quantbody ")" "-" Const

Quantbody \rightarrow Var ":" "(" Ivar ".." Ivar ")" ":"

Avar "[" Var "]" Relop Const

Lexp0 \rightarrow "(" Lexp0 ")"

Lexp0 \rightarrow Lexp1 Lexp0more

Lexp1 \rightarrow "(" Lexp1 ")"

Lexp1 \rightarrow Lexp2 Lexp1more

Lexp2 \rightarrow "(" Lexp2 ")"

Lexp2 \rightarrow Lexp3 Lexp2more

Lexp3 \rightarrow "(" Lexp3 ")"

Lexp3 \rightarrow Lexp4 Lexp3more

Lexp4 \rightarrow "(" Lexp4 ")"

Lexp4 \rightarrow Rexp Lexp4more

Lexp0more \rightarrow "eqv" Lexp0

Lexp0more \rightarrow null

Lexp1more \rightarrow "imp" Lexp1

Lexp1more \rightarrow null

Lexp2more \rightarrow "xor" Lexp2

Lexp2more \rightarrow null

```

Lexp3more -> "or" Lexp3
Lexp3more -> null

Lexp4more -> "and" Lexp4
Lexp4more -> null

Rexp -> "not" Rexp
Rexp -> Exp Relop Exp

Exp -> Const Constmore
Exp -> Avar "[" Exp "]"

Constmore -> "+" Exp
Constmore -> "-" Exp
Constmore -> null

Relop -> "=" | ">" | "<" | "<=" | ">=" | "<" | ">"
Lop -> "and" | "or" | "xor" | "imp" | "eqv"

Const -> int {any legal integer}
Const -> Var

Ivar -> ivar {any defined index variable}
Var -> var {any defined variable}
Avar -> avar {any defined array variable}

```

Definite Clause Grammars

```
interp(L,Val) -> pred(L1,V1), morepred(L2,V2,Op),
    { L2=[], L=L1, Val=V1;
      L=lexp(Op,L1,L2), Val=V1<Op>V2 }.

morepred(L,Val,Op) -> [], {L=[]}.
morepred(L,Val,Op) -> lop(Lop), pred(L1,V1), morepred(L2,V2,Op1),
    { L2=[], Op=Lop, L=L1, Val=V1;
      L=lexp(Op1,L1,L2), Val=V1<Op1>V2, Op=Lop }.

pred(L,Val) -> [], quant(L,Val), [].
pred(L,Val) -> lexp0(L,Val).

lexp0(L,Val) -> [], lexp0(L,Val), [].
lexp0(L,Val) -> lexp1(L1,V1), lexpmore(eqv,L1,V1,L,Val).

lexp1(L,Val) -> [], lexp1(L,Val), [].
lexp1(L,Val) -> lexp2(L1,V1), lexpmore(imp,L1,V1,L,Val).

lexp2(L,Val) -> [], lexp2(L,Val), [].
lexp2(L,Val) -> lexp3(L1,V1), lexpmore(xor,L1,V1,L,Val).

lexp3(L,Val) -> [], lexp3(L,Val), [].
lexp3(L,Val) -> lexp4(L1,V1), lexpmore(or,L1,V1,L,Val).

lexp4(L,Val) -> [], lexp4(L,Val), [].
lexp4(L,Val) -> rexp(L1,V1), lexpmore(and,L1,V1,L,Val).

lexpmore(eqv,L1,V1,L,Val) -> [eqv], lexp0(L2,V2),
    { Val= evaluated result from "V1 'eqv' V2",
      L=lexp([eqv,L1,L2]) }.
lexpmore(imp,L1,V1,L,Val) -> [imp], exp1(L2,V2),
    { Val= evaluated result from "V1 'imp' V2",
      L=lexp([imp,L1,L2]) }.
lexpmore(xor,L1,V1,L,Val) -> [xor], lexp2(L2,V2),
    { Val= evaluated result from "V1 'xor' V2",
      L=lexp([xor,L1,L2]) }.
lexpmore(or,L1,V1,L,Val) -> [or], lexp3(L2,V2),
    { Val= evaluated result from "V1 'or' V2",
      L=lexp([or,L1,L2]) }.
```

```

lexpmore(and,L1,V1,L,Val) -> [and], lexp4(L2,V2),
    { Val= evaluated result from "V1 'and' V2",
      L=lexp([and,L1,L2]) }.
lexpmore(Lop,L1,V1,Lf,Vf) -> [],
    { Lop="any defined logical operator", Lf=L1, Vf=V1 }.

quant(L,Val) -> [forall], quantbody(L1,Val), { L=forall(L1) }.
quant(L,Val) -> [thereexist], quantbody(L1,Val), { L=thereexist(L) }.
quant(L,Val) -> [], [numberof], quantbody(L1,Val), [], [=], const.

quantbody -> var, [], [], ivar, [...], ivar, [], [],
    avar, [T], var, [T], relop, const.

rexp(L,Val) -> [not], rexp(L1,V1), { Val= negation of value V1, L=not(L1) }.
rexp(L,Val) -> exp(L1,V1), relop(Rop), exp(L2,V2),
    { Val= evaluated result from "V1 Rop V2",
      L=rexp([Rop,L1,L2]) }.

exp(L,Val) -> const(L1,V1), moreconst(L2,V2,Op),
    { Val= "V1 Op V2", L=[Op,L1,L2] }.
exp(L,Val) -> [Avar], [T], exp(Ivar,V1), [T],
    { Va= the attribute of "Avar[V1]",
      L=[array.avar(Avar),ivar(Ivar)] }.

relop(>=) -> [>], [=].
relop(<=) -> [<], [=].
relop(<>) -> [<], [>].
relop(=) -> [=].
relop(<) -> [<].
relop(>) -> [>].

lop(and) -> [and].
lop(or) -> [or].
lop(xor) -> [xor].
lop(imp) -> [imp].
lop(eqv) -> [eqv].

```

```

moreconst([],[],[]) -> []
moreconst(L,Val, '+') -> [+] exp(L,Val)
moreconst(L,Val, '-') -> [-] exp(L,Val)

const(num(X)) -> [X], { X is a legal integer }.
const(var(X)) -> [X], { X == the attribute of val(X) }.
const(ivar(X)) -> [X], { X == any ivar(X) defined in the database }.

```


Appendix II: C-Prolog Implementations

```
% *****
% "main.pro"
% *****
%
% This program performs the following functions:
%   the files consultation, the creation/control of menu,
%   the linkage to scanner and parser programs, and
%   the output of the results.
% -----

main :- consult('pred.scan.pro'),
        consult('pred.parse.pro'),
        consult('pred.db.pro'),
        assertdb,
        nl, menu.

menu :- write('          \\\\\\\\\\\ MENU \\\\\\\\\\\ '), nl,
        write('----- '), nl,
        write(' a) simple predicate. '), nl,
        write(' b) predicate with universal quantification. '), nl,
        write(' c) predicate with existential quantification. '), nl,
        write(' d) predicate with numerical quantification. '), nl,
        write(' o) user input.i '), nl,
        write(' q) quit from the program. '), nl,
        write(' Please enter the code, then hit <cr> '), nl,
        get(Code), get0(_),
        testfile(Code,Repeat),
        Repeat=true, menu, nl.

menu.

testfile(C,true) :- C=97, start('test.simple.pred',user).
testfile(C,true) :- C=98, start('test.qall',user).
testfile(C,true) :- C=99, start('test.qexist',user).
testfile(C,true) :- C=100, start('test.qnum',user).
testfile(C,false) :- C=113, retractdb, !.
testfile(C,true) :- C=111,
        write('please enter the file name, followed by '),
        write('a dot and <cr> '), nl,
        read(File), nl, start(File,user).
```

```

testfile(____,Val) :- nl,
                      write('Please re-enter the code, then hit <cr> '),
                      nl, get(Code), get0(____),
                      testfile(Code,Val).

start(Infile, Outfile) :-
    see(Infile),
    tell(Outfile),
    write(' Input tokens '), nl,
    write(' ----- '), nl,
    get0(Ch),
    scan(Ch,Wordslist),
    seen, nl, nl,
    write(' Parsing/Evaluation '), nl,
    write(' ----- '), nl,
    interpret(Parselist,Val,Wordslist),
    write(Parselist), nl,
    write(' ** the result ==> '),
    write(Val), nl, nl, nl,
    told.
start(____) :- seen, told.

```

```

% *****
% "pred.db.pro"
% *****
%
% The information stored in this database includes:
% integer variables (ivar)
%     m = 0, n = 3,
%     x = 1, y = 3, z = 5,
%     a = 10, b = 20, c = 30.
%
% array variables (avar)
%     byte = "00001111"
%     count = "0123456789"
% -----

assertdb :- db, valdb, bindb, countdb, ivardb.

retractdb :- db, valout, binout, countout.
retractdb :- nl,
    write(' ** empty database ** '),
    nl.

db :- assert(avar(byte)), assert(avar(count)).

ivardb :- assert(ivar(m,0)), assert(ivar(n,3)).
ivarout :- retract(ivar(_,_)), ivarout.
ivarout.

valdb :- assert(val(x,1)), assert(val(y,3)), assert(val(z,5)),
    assert(val(a,10)), assert(val(b,20)), assert(val(c,30)).
valout :- retract(val(_,_)), valout.
valout.

% Arrays
% -----

bindb :-
    assert(byte(0,1)), assert(byte(1,1)), assert(byte(2,1)),
    assert(byte(3,1)), assert(byte(4,0)), assert(byte(5,0)),
    assert(byte(6,0)), assert(byte(7,0)).

binout :- retract(byte(_,_)), valout.
binout.

```

```
countdb :-  
    assert(count(0,0)), assert(count(1,1)), assert(count(2,2)),  
    assert(count(3,3)), assert(count(4,4)), assert(count(5,5)),  
    assert(count(6,6)), assert(count(7,7)), assert(count(8,8)),  
    assert(count(9,9)).
```

```
countout :- retract(count(_,_)), countout.  
countout.
```

```

% *****
% "pred.scan.pro"
% *****
%
% This program performs the lexical analysis of
% the input.
% -----

scan(26,[]) :- !, % "Z" ==> end of input
scan(32,TokenList) :-
    get0(Ch), !,
    scan(Ch,TokenList).
scan(10,TokenList) :-
    get0(Ch), !,
    scan(Ch,TokenList).
scan(Ch,[Wd/TokenList]) :-
    getspecialsymbol(Ch,Wd),
    write(Wd), tab(1),
    get0(Ch1), !,
    scan(Ch1,TokenList).
scan(Ch,[Wd/TokenList]) :- % get a word or a number
    getword(Ch,Wd1.LastCh),
    name(Wd,Wd1),
    write(Wd), tab(1), !,
    scan(LastCh,TokenList).
scan(Ch,[Wd/TokenList]) :-
    name(Wd,[Ch]), nl,
    write(Wd),
    write(' <== ** invalid token **'),
    nl, nl,
    !, fail.

getword(Ch,[Ch/Wd1],LastCh) :-
    letter(Ch), !,
    get0(Ch1),
    isname(Ch1,Wd1.LastCh).
getword(Ch,[Ch/Wd1],LastCh) :-
    digit(Ch), !,
    get0(Ch1),
    isnumber(Ch1,Wd1.LastCh).

```

```

isname(Ch,[Ch/Wd],LastCh) :-
    letter(Ch), !,
    get0(Ch1), isname(Ch1,Wd,LastCh).
isname(Ch,[Ch/Wd],LastCh) :-
    digit(Ch), !,
    get0(Ch1), isname(Ch1,Wd,LastCh).
isname(Ch,[],LastCh) :-
    LastCh=Ch.

isnumber(Ch,[Ch/Wd],LastCh) :-
    digit(Ch), !,
    get0(Ch1), isnumber(Ch1,Wd,LastCh).
isnumber(Ch,[],LastCh) :-
    LastCh=Ch.

letter(Ch) :- Ch>64, Ch<91;    % capital
              Ch>96, Ch<123.   % small letter

digit(Ch) :- Ch>47, Ch<58.

```

% Operators

% _____

```

getspecialsymbol(40,'(').
getspecialsymbol(41,')').
getspecialsymbol(61,'=').
getspecialsymbol(91,'[').
getspecialsymbol(93,']').
getspecialsymbol(60,'<').
getspecialsymbol(62,'>').
getspecialsymbol(46,',').
getspecialsymbol(58,':').
getspecialsymbol(43,'+').
getspecialsymbol(45,'-').

```

% print facility

% _____

```

writeall([]) :- nl, nl, !.
writeall([X:X1]) :-
    write(" "), write(X), write(" "),
    tab(2), writeall(X1).

```

```
% *****
% "pred.parse.pro"
% *****
%
% This program performs the functions of parsing and execution.
% -----
%
% The precedence of the logic operators is and> or> xor> imp> eqv.
% and it has been implemented into the source grammar. However,
% it is recommended that user always use parantheses for input
% expression.
%
% All the quantification SHOULD BE PARENTHESESIZED.
%   For example:
%       (thereexist i:(j..k):count[i]=4)
%       (forall i:(j..k):byte[i]=0)
%       ((numberof i:(j..k):byte[i]=1)=4)
% -----
```

```
interpret(ParseList,Val,TokenList) :- interp(ParseList,Val,TokenList,[]).
interpret(ParseList,Val,TokenList) :- told, t, fail.
```

```
% *****
% Parsing with the Definite Clause Grammar Rule
% *****
```

```
interp(L,Val) --> pred(L1,V1), morepred(L2,V2,Op),
    { L2=[], L=L1, Val=V1;
      L=lexp([Op,L1,L2]), Val=V, evallog(V,V1,V2,Op) }.
```

```
pred(L,Val) --> ['(', quant(L,Val), [')'].
pred(L,Val) --> lexp0(L,Val).
```

```
morepred(L,_) --> [], {L=[]}.
morepred(L,Val,Op) -->
    lop(Op), pred(L1,V1), morepred(L2,V2,Op1),
    { L2=[], L=L1, Val=V1;
      L=lexp([Op1,L1,L2]), evallog(V,V1,V2,Op1), Val=V }.
```

```

quant(forall(Rel,A,C),Val) ->
  [forall], quantbody(Avar,Id1,Id2,Rel,C),
  { universal(Avar,Id1,Id2,Rel,C,V),
    Val=V, A=..[Avar,Id1,Id2]    },
quant(thereexist(Rel,A,C),Val) ->
  [thereexist], quantbody(Avar,Id1,Id2,Rel,C),
  { exist(Avar,Id1,Id2,Rel,C,V),
    Val=V, A=..[Avar,Id1,Id2]    },
quant(numberof(=,[Rel,A,C],Count),Val) ->
  ['('], [numberof], quantbody(Avar,Id1,Id2,Rel,C), [')'],
  ['='], const(_Count),
  { numerical(Avar,Id1,Id2,Rel,C,0,Cnum),
    Cnum=Count, Val=true, A=..[Avar,Id1,Id2], !;
    Val=false
  },

quantbody(Avar,Id1,Id2,Relop,Const) ->
  [X1], ['('], ['('], [I1], ['('], ['('], [I2], [')'], ['('], ['('],
  [Av], ['('], [X2], [')'], relop(Relop), const(_Const),
  { X1=X2, avar(Av), Avar=Av,
    const(_J1,[I1],[]),
    const(_J2,[I2],[]),
    J1 < J2, Id1=J1, Id2=J2    },

lexp0(L,Val) -> ['('], lexp0(L,Val), [')'],
lexp0(L,Val) -> lexp1(L1,V1), lexpmore(eqv,L1,V1,L,Val),

lexp1(L,Val) -> ['('], lexp1(L,Val), [')'],
lexp1(L,Val) -> lexp2(L1,V1), lexpmore(imp,L1,V1,L,Val),

lexp2(L,Val) -> ['('], lexp2(L,Val), [')'],
lexp2(L,Val) -> lexp3(L1,V1), lexpmore(xor,L1,V1,L,Val),

lexp3(L,Val) -> ['('], lexp3(L,Val), [')'],
lexp3(L,Val) -> lexp4(L1,V1), lexpmore(or,L1,V1,L,Val),

lexp4(L,Val) -> ['('], lexp4(L,Val), [')'],
lexp4(L,Val) -> rexp(L1,V1), lexpmore(and,L1,V1,L,Val),

```



```

lexpmore(eqv,L1,V1,Lf,Vf) ->
    [eqv], lexp0(L2,V2),
    { evallog(V,V1,V2,eqv), Vf=V, Lf=lexp([eqv,L1,L2]) }.
lexpmore(imp,L1,V1,Lf,Vf) ->
    [imp], lexp1(L2,V2),
    { evallog(V,V1,V2,imp), Vf=V, Lf=lexp([imp,L1,L2]) }.
lexpmore(xor,L1,V1,Lf,Vf) ->
    [xor], lexp2(L2,V2),
    { evallog(V,V1,V2,xor), Vf=V, Lf=lexp([xor,L1,L2]) }.
lexpmore(or,L1,V1,Lf,Vf) ->
    [or], lexp3(L2,V2),
    { evallog(V,V1,V2,or), Vf=V, Lf=lexp([or,L1,L2]) }.
lexpmore(and,L1,V1,Lf,Vf) ->
    [and], lexp4(L2,V2),
    { evallog(V,V1,V2,and), Vf=V, Lf=lexp([and,L1,L2]) }.
lexpmore(_L1,V1,Lf,Vf) -> [], { Lf=L1, Vf=V1 }.

```

```

rexp(not(L),Val) ->
    [not], rexp(L,V), { Val=true, V=false: Val=false }.
rexp(rexp([Op,L1,L2],Val) ->
    exp(L1,V1), relop(Op), exp(L2,V2),
    { Val=V, evalrel(V,V1,V2,Op) }.

```

```

exp(L,Val) -> const(L,Val).
exp([plus,L1,L2],Val) ->
    const(L1,V1), ['+'], exp(L2,V2), { Val is V1+V2 }.
exp([minus,L1,L2],Val) ->
    const(L1,V1), ['-'], exp(L2,V2), { Val is V1-V2 }.
exp([array,avar(X),ivar(L)],Val) ->
    [X], ['[', exp(L,V1), ']'],
    { avar(X), Val=V2, Goal =.. [X,V1,V2], call(Goal) }.

```

```

relop('>=') -> ['>'], ['='].
relop('<>') -> ['<'], ['>'].
relop('<=') -> ['<'], ['='].
relop('>') -> ['>'].
relop('=') -> ['='].
relop('<') -> ['<'].

```

```

lop('and') -> [and].
lop('or') -> [or].
lop('xor') -> [xor].
lop('imp') -> [imp].
lop('eqv') -> [eqv].

```

```

const( num(X),X) -> [X], {number(X)}.
const( var(X),Y) -> [X], {val(X,Y1), Y=Y1}.
const(ivar(X),Y) -> [X], {ivar(X,Y1), Y=Y1}.

```

```

% *****
% The following predicates are the user-defined functions.
% Each predicate is invoked by a procedure call from the
% extra condition (semantic actions) in the rules above.
% *****

```

```

% Evaluation of the values specified in quantifications
% -----

```

```

universal(_Low,High,_,true) :- Low>High.
universal(Avar,Low,High,Op,Const,Val) :-
    Goal1 =.. [Avar,Low,Num], call(Goal1),
    Goal2 =.. [evalrel,V2,Num,Const,Op], call(Goal2),
    V2=true, Next is Low + 1,
    universal(Avar,Next,High,Op,Const,Val).
universal(____,false).

```

```

exist(_Low,High,_,false) :- Low>High.
exist(Avar,Low,High,Op,Const,Val) :-
    Goal1 =.. [Avar,Low,Num], call(Goal1),
    Goal2 =.. [evalrel,V2,Num,Const,Op], call(Goal2),
    V2=false, Next is Low + 1,
    exist(Avar,Next,High,Op,Const,Val).
exist(____,true).

```

```

numerical(,Low,High,Count,Count) :- Low>High.
numerical(Avar,Low,High,Op,Const,Count,Total) :-
    Goal1 =, [Avar,Low,Num], call(Goal1),
    Goal2 =, [evalrel,V2,Num,Const,Op], call(Goal2),
    V2=true, Cnum is Count + 1, Next is Low + 1,
    numerical(Avar,Next,High,Op,Const,Cnum,Total).
numerical(Avar,Low,High,Op,Const,Count,Total) :-
    Next is Low + 1,
    numerical(Avar,Next,High,Op,Const,Count,Total).

```

```

% Evaluation of a logical expression
% Note: (V1 imp V2) == (not(V1) or V2)
% -----

```

```

evallog(true,V1,V2,and) :- V1=V2, V1=true, !.
evallog(true,V1,V2,or) :- not((V1=V2,V1=false)), !.
evallog(true,V1,V2,xor) :- not(V1=V2), !.
evallog(true,V1,V2,imp) :- not((V1=true,V2=false)), !.
evallog(true,V1,V2,eqv) :- V1=V2, !.
evallog(false,V1,V2,Op).

```

```

% evaluation of a relational expression
% -----

```

```

evalrel(true,X,Y,'>') :- X>Y, !.
evalrel(true,X,Y,'=') :- X=Y, !.
evalrel(true,X,Y,'<') :- X<Y, !.
evalrel(true,X,Y,'<=>') :- X==Y, !.
evalrel(true,X,Y,'<=') :- X=<Y, !.
evalrel(true,X,Y,'>=') :- X>=Y, !.
evalrel(false,X,Y,Op).

```

Appendix III: Turbo Prolog implementations

```

/*      *main.pro*      */
/*      -----      */
/*      */
/* This program covers the functions of */
/* main control and parsing/execution */
/* -----      */

code=2048

domains
    tok = int(integer); op1(char); op(string); name(string)
    tok1 = tok*
    values = symbol*

database
    val(string,integer)      /* variable */
    avar(string)             /* array variable */
    lvar(string,integer)     /* index variable */
    array(string,integer,integer) /* social security no. */

    valdb(values)           /* stack for boolean values */
    lopdb(tok1)             /* stack for logical operators */
    preddb(string,integer)  /* for debugging use */

include "database.inc"
include "stacks.inc"
include "scanner.inc"
include "operators.inc"

predicates
    start
    writeln(string)
    pred(tok1,symbol,tok1)
    lexp(tok1,symbol,tok1)
    xexp(tok1,tok1,tok1,integer)

    check(tok1)

```

```

morelexp(tok1,tok1)
openp(tok)

negate(symbol,symbol)
evalop(tok)
fineval(symbol,symbol)
range(integer,integer,integer,symbol)
frontlist(tok1,tok,tok1)
quant(symbol,string,integer,integer,integer,
        integer,symbol,integer,integer,integer)

typeval(symbol)
writeval(tok,symbol,symbol,symbol)

goal
    start.

clauses
    writeln(X) :- write(X), nl.

/* main control and utilities */
start :-
    writeln("simple predicate interpreter"),
    writeln("enter expression"),
    readln(Str),nl, !,
    scan(Str,Tok1),
    init,
    check(Tok1),
    reinit,
    writeln("done").

check(Tok1) :-
    pred(Tok1,V,[],) , nl, write("result = ").
    typeval(V), writeln("*** pass check ***"), !.
check(_) :- nl, writeln("*** fail pass ***").

```

```
/*      more printer facilities
----- */
```

```
typeval(true) :- writeln("true"), !.
typeval(_) :- writeln("false").
```

```
writeval(op("and"),V1,V2,Vf) :-
    write(V1), write(" <and> "), write(V2),
    write(" ==> "), write(Vf), nl.
writeval(op(Lop),V1,V2,Vf) :-
    write(V1),write(" <"),write(Lop),write(" > "),
    write(V2),write(" ==> "),write(Vf),nl, !.
```

```
/*      parser
----- */
```

```
pred([op("thereexist")/Tok1],Vpred,Tok) :-
    frontlist(Tok1.name(Indx),[op1('/:')/Tok2]],
    frontlist(Tok2.op1('['),[name(X1)/Tok3]],
    ivar(X1,V1),
    frontlist(Tok3.op(".."),[name(X2)/Tok4]],
    ivar(X2,V2),
    V2>V1,
    frontlist(Tok4.op1(')'),[op1('/:')/Tok5]],
    frontlist(Tok5.name(Avar),[op1('[')/Tok6]],
    avar(Avar),
    frontlist(Tok6.name(Indx),[op1('/:')/Tok7]],
    frontlist(Tok7.op1('='),Tok8),
    sexp(Tok8,_,Tok.Data),
    quant((thereexist,Avar,V1,V2,_,Data,Vpred,_,_)),
    !.
pred(Tok1,Vprops,Tok2) :-
    lexp(Tok1,Vlexp,Retoks),
    pushval(Vlexp),
    morelexp(Retoks,Tok2),
    popval(V2),
    fineval(V2,Vprops), !.
```

```

quant(thereexist,A,L,Data,true,_) :-
    array(A,L,Data), !.
quant(thereexist,A,L,H,Next,Data,Val,_) :-
    Next=L+1,
    H >= Next,
    quant(thereexist,A,Next,H,N,Data,Val,_), !.
quant(thereexist,_,_,_,false,_) :- !.

```

```

morelexp([],[]) :- !.
morelexp([op1(T)(R1)], [op1(T)(R2)] :-
    T=')', !, R2=R1, !.
morelexp([TffT],Tok2) :-
    lop(Tf), pushlop(Tf),
    lexp(T1,V,Retoks), !,
    pushval(V),
    poplop(Op),
    evalop(Op),
    morelexp(Retoks,Tok2), !.

```

```

lexp([op1(T)(Retok1),V,Retok2) :-
    T='(',
    pushlop(op1(T)),
    pred(Retok1,V,[op1(T1)(Retok2)],
    T1=')',
    poplop(_), !.
lexp([op(T)(Retok1),V,Retok2) :-
    T="not",
    pushlop(op(T)),
    lexp(Retok1,V1,Retok2),
    negate(V,V1),
    poplop(op(T)), !.
lexp([name(X)(T),Val,Rtoks) :-
    ivar(X,V), !,
    frontlist(T,op("in"),[op1('')(T1)],
    frontlist(T1.name(X1),[op("")(T2)],
    ivar(X1,V1), !,
    frontlist(T2.name(X2),[op1('')(Rtoks)],
    ivar(X2,V2), !,
    range(V,V1,V2,Val), !.
lexp(T,Val,Rtoks) :-
    sexp(T,_,[T1/T2],V1), !,
    relop(T1),
    sexp(T2,_,Rtoks,V2),
    evalrop(T1.V1,V2,Val), !.

```

```

evalop(op("not")) :-
    popval(V),
    negate(Vf,V),
    pushval(Vf), !.
evalop(op("and")) :-
    popval(V), popval(V1),
    evallop(op("and"),V1,V,Vf),
    writeval(op("and"),V1,V,Vf),
    pushval(Vf), !.
evalop(L2) :-
    poplop(L1),
    preced(L2,L1), !.
    popval(V), popval(V2), popval(V1),
    evallop(L1,V1,V2,Vfinal),
    writeval(L1,V1,V2,Vfinal),
    pushval(Vfinal), pushval(V),
    evalop(L2), !.
evalop(L2) :-
    pushlop(L2), !.

fineval(V2,Val) :-
    poplop(Lop),
    pushlop(Lop),
    openp(Lop), !,
    Val=V2.
fineval(V2,Val) :-
    popval(V1), poplop(Lop),
    evallop(Lop,V1,V2,Vf),
    writeval(Lop,V1,V2,Vf),
    fineval(Vf,Val), !.
fineval(V2,V2) :- !.

sexp([name(X)T2],T2,T2,V) :- val(X,V), !.
sexp([name(X)T2],T2,Tf,V) :-
    avar(X), !,
    sexp(T2,_,Tf,V1),
    array(X,V1,V), !.
sexp([int(N)T2],T2,T2,V) :- V = N, !.
sexp([op1(')T2],T2,Tf,V) :-
    sexp(T2,[op1(')Tf],_,V), !.

```



```
negate(Vn,true) :- !, Vn=false.
negate(Vn,_)   :- Vn=true, !.
```

```
frontlist([T1/T2],T1,T2) :- !.
```

```
range(N,N1,N2,true) :-
    N >= N1, N <= N2, !.
range(_,_,false) :- !.
```

```
openp(op1(T)) :- T='(', !.
```

```

/*      "database.inc"      */
/*      =====      */

```

```

predicates
  init
  reinit
  ssninit

```

```

clauses

```

```

init :-
  assert(ivar("one",1)), assert(ivar("two",2)),
  assert(ivar("three",3)), assert(ivar("four",4)),
  assert(ivar("five",5)), assert(ivar("six",6)),
  assert(ivar("seven",7)), assert(ivar("eight",8)),
  assert(ivar("nine",9)), assert(ivar("ten",10)),
  assert(ivar("eleven",11)), assert(ivar("twelve",12)),
  assert(ivar("thirteen",13)), assert(ivar("fourteen",14)),
  assert(ivar("fifteen",15)),

  assert(avar("ssn")),
  assert(array("ssn",1,5)), assert(array("ssn",2,1)),
  assert(array("ssn",3,5)), assert(array("ssn",4,8)),
  assert(array("ssn",5,2)), assert(array("ssn",6,8)),
  assert(array("ssn",7,4)), assert(array("ssn",8,0)),
  assert(array("ssn",9,3)),

  assert(val("x",3)), assert(val("y",2)), assert(val("z",1)),
  assert(valdb([])), assert(lopdb([])),
  assert(predb("props ",0)), assert(predb("molexp",0)),
  assert(predb("lexp ",0)), assert(predb("evalop",0)).

reinit :-
  retract(val("x",_)), retract(val("y",_)), retract(val("z",_)),
  retract(valdb(_)), retract(lopdb(_)),
  retract(predb("props ",_)), retract(predb("molexp",_)),
  retract(predb("lexp ",_)), retract(predb("evalop",_)),
  ssninit.

```

reinit :-

```
retract(val("x",_)), retract(val("y",_)), retract(val("z",_)),  
retract(valdb(_)), retract(lopdb(_)),  
retract(predb("props",_)), retract(predb("molexp",_)),  
retract(predb("lexp",_)), retract(predb("evalop",_)),  
ssninit.
```

ssninit :-

```
retract(array("ssn",1,_)), retract(array("ssn",2,_)),  
retract(array("ssn",3,_)), retract(array("ssn",4,_)),  
retract(array("ssn",5,_)), retract(array("ssn",6,_)),  
retract(array("ssn",7,_)), retract(array("ssn",8,_)),  
retract(array("ssn",9,_)).
```

```
/*      "stacks.inc"
```

```
-----  
This programs performs the function of a stack
```

```
*/
```

```
predicates
```

```
    pushval(symbol)  
    popval(symbol)  
    pushlop(tok)  
    poplop(tok)
```

```
clauses
```

```
    popval(V) :-  
        retract(valdb([V|Vt])),  
        assert(valdb(Vt)),!,  
    pushval(V) :-  
        retract(valdb(Vold)),  
        assert(valdb([V|Vold])),!,  
  
    poplop(L) :-  
        retract(lopdb([L|Lt])),  
        assert(lopdb(Lt)),!,  
    pushlop(L) :-  
        retract(lopdb(Lold)),  
        assert(lopdb([L|Lold])),!.
```

```
/*      "scanner.inc"      */
/*      =====      */
```

predicates

```
scan(string,tok1)
makeop(string,char,tok)
maketok(string,tok)
isres(string)
```

clauses

```
scan("",[]) :- !.
scan(Str,[Tok|Tok1]) :-
    fronttoken(Str,C1,Str2),
    frontchar(Str2,C2,Str3),
    makeop(C1,C2,Tok),
    scan(Str3,Tok1).
```

```
scan(Str,[Tok|Tok1]) :-
    fronttoken(Str,Sym,Str2),
    maketok(Sym,Tok),
    scan(Str2,Tok1).
```

```
makeop(">", '-', op(">=")).
makeop("<", '-', op("<=")).
makeop("<", '>', op("<>")).
makeop("=", ':', op("=..")).
```

```
maketok(S,op(S)) :- isres(S), !.
maketok(S,name(S)) :- isname(S), !.
maketok(S,int(N)) :- str_int(S,N), !.
maketok(S,op1(C)) :- str_char(S,C), !.
```

```
isres(S) :-
    S = "in"; S = "and"; S = "or";
    S = "xor"; S = "eqv"; S = "imp";
    S = "not"; S = "thereexist";
    S = "forall"; S = "numberof".
```

```

/*      "operators.inc"      */
/*      -----      */

```

```

predicates
    lop(tok)
    relop(tok)

    evalrop(tok.integer.integer.symbol)
    preced(tok,tok)
    evallop(tok.symbol.symbol.symbol)

```

clauses

```

/*      Determination of the precedence order      */
/*      between two logical operators.      */
/*      -----      */

```

```

preced(op(T),op("or")) :-
    T = "or", !;
    T = "xor", !;
    T = "imp", !;
    T = "eqv", !;
    pushlop(op("or")), !, fail.
preced(op(T),op("xor")) :-
    T = "xor", !;
    T = "imp", !;
    T = "eqv", !;
    pushlop(op("xor")), !, fail.
preced(op(T),op("imp")) :-
    T = "imp", !;
    T = "eqv", !;
    pushlop(op("imp")), !, fail.
preced(op(T),op("eqv")) :-
    T = "eqv", !;
    pushlop(op("eqv")), !, fail.
preced(op(____),T) :-
    pushlop(T), !, fail.      /* save op(') */

```

```

/*  Evaluation of logical expression      */
/*  -----                               */

```

```

evallop(op("and"),V1,V2,Vf) :-
    V1 = true, V2 = V1, Vf = V1;
    Vf = false.
evallop(op("or"),V1,V2,Vf) :-
    V1 = true, Vf = V1;
    V2 = true, Vf = V2;
    Vf = false.
evallop(op("xor"),V1,V2,Vf) :-
    V1 <> V2, Vf = true;
    Vf = false.
evallop(op("imp"),V1,V2,Vf) :-
    V1 = true, Vf = V1;
    V2 = V1, Vf = true;
    Vf = false.
evallop(op("eqv"),V1,V2,Vf) :-
    V1 = V2, Vf = true;
    Vf = false.

```

```

/*  Categories of operators              */
/*  -----                               */

```

```

relop(op1(T)) :- T = '>', !; T = '<', !; T = '=', !.
relop(op(T)) :- T = ">=", !; T = "<=", !; T = "<>".
lop(op(T)) :- T = "and", !; T = "or", !; T = "xor", !;
    T = "imp", !; T = "eqv".

```

```

/*  evaluation of relational expression  */
/*  -----                               */

```

```

evalrop(op1(P),V1,V2,true) :- P = '>', V1 > V2.
evalrop(op1(P),V1,V2,true) :- P = '<', V1 < V2.
evalrop(op1(P),V1,V2,true) :- P = '=', V1 = V2.
evalrop(op1(____),Val) :- Val = false, !.
evalrop(op(P),V1,V2,true) :- P = ">=", V1 >= V2.
evalrop(op(P),V1,V2,true) :- P = "<=", V1 <= V2.
evalrop(op(P),V1,V2,true) :- P = "<>", V1 <> V2.
evalrop(op(____),false).

```

Appendix IV: Test Files

```
% "test.simple.pred"
% _____
```

$(x=1) \text{ or } (x=3) \text{ and } (z=y+3) \wedge Z$

```
% "test.gall"
% _____
```

$(\text{forall } i : (m..n) : \text{byte}[i]=1) \wedge Z$

```
% "test.qexist"
% _____
```

$(\text{therexist } i : (2..6) : \text{count}[i]=5) \wedge Z$

```
% "test.qnum"
% _____
```

$((\text{numberof } i : (0..7) : \text{byte}[i]=1) = 4) \wedge Z$

```
% "test.user"
% _____
```

$(x=1 \text{ and } \text{byte}[3]=1) \text{ or } (\text{therexist } i : (1..4) : \text{byte}[i]=1) \text{ and } x > y \wedge Z$

Appendix V: Formal Grammar for Definite Clause Grammars

```

dcg ::= lhs '→' rhs

lhs ::= rule_head

rhs ::= parse_phrase [' more_rhs]

rule_head ::= pred_term

parse_phrase ::= [const_term | pred_term]#

pred_term ::= string ['(' arg_list ')']

more_rhs ::= extra_cond [' rhs]*

extra_cond ::= '[' semantic_rule '[' semantic_rule]* '['

semantic_rule ::= users_proc_call | system_call

users_proc_call ::= procedure_name ['(' arg_list ')']

procedure_name ::= { a procedure_name is the name of a predicate which
                      is defined in the program by the user }

system_call ::= { a system call can be any built-in function described
                  in the C-Prolog user's manual }

const_term ::= '[' [const | var] '['

compd_term ::= (string | var) ['(' arg_list ')']

arg_list ::= argument [' argument]

argument ::= list | term

list ::= '[' term [term1 | term2] '['

term ::= var | const | compd_term

term1 ::= '!' term

term2 ::= '[' term]#

```

```

var ::= cap_letter [alphabet | digit]
const ::= string | numeral | atom
string ::= small_letter [alphabet | digit]
alphabet ::= small_letter | cap_letter
numeral ::= [digit]#
small_letter ::= 'a' 'b' 'c' ... 'x' 'y' 'z'
cap_letter ::= 'A' 'B' 'C' ... 'X' 'Y' 'Z'
digit ::= '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'
atom ::= { any printable character }

```

Notes:

- I. The following are the interpretations for some notations used in the BNF:
 - 1) A small-letters string is a non_terminal, such as "dcg".
 - 2) A term within a pair of "" is a terminal, where a terminal can be a letter, a constant or an atom, such as '0', '{', and '->'.
 - 3) The repetition of terms can be expressed in several ways:
 - (term1term2) == either term1 or term2 occurs, and it does once.
 - [term] == term occurs at most once.
 - [term]# == term occurs at least once.
 - [term]* == term occurs zero or more times.
- II. The non-terminal "system_call" can be any built-in function defined in C-Prolog user's manual (Version 1.4).

USE OF DEFINITE CLAUSE GRAMMARS

by

Weilin T. Chen

M.S. Kansas State University, KS, 1985

B.S. Providence College, Taiwan, 1980

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Abstract

Prolog is an excellent language for developing parsers because Prolog rules are similar to BNF rules and the Prolog execution algorithm is similar to the LL(1) parsing algorithm. This report presents a study of Definite Clause Grammars for the parsing using Prolog. Definite Clause Grammars (DCGs) are an extension of Context Free Grammars (CFGs).

This report presents a tutorial on the use of Definite Clause Grammars. Also, a formal grammar for DCGs is given. As a sample problem, a small language interpreter is implemented with the use of a DCG is written in C-Prolog. For comparison, another version without the use of a DCG is implemented in Turbo Prolog. The two implementations are compared first on syntactical measurement and second on subjective evaluation. The syntactical measurement consists of counts of the constituents for each predicate. The subjective evaluation describes the degree of user effort required for the program design and coding, and the overall readability of the resulting program.

From the aspect of implementation design, the use of a DCG for Prolog parsing is preferred despite the possibility of higher syntactical complexity.